Big Data Approach to fMRI Data Analysis with Intel DAAL and Full Correlation Matrix Analysis

Yin Lok Wong

Abstract—**This report describes the combination of the use and testing of the open source Spark NIfTI reader library biananes,[1] dedicated for scalable fMRI data analysis, together with Intel Data Analytics Acceleration Library(DAAL), to realize analytic operations of fMRI data on the Big Data framework Apache Spark and replacing the use of Apache Spark MLlib with DAAL under the Computational Science for Undergraduate Research Experience programme(CSURE 2016).**

I. Introduction

fMRI data analysis deals with data represented in large scale matrices. Operations on those matrices generate intermediate and resulting matrices and data that are often even larger, for example, the use of data in gigabytes might generate results in terabytes, which breeds an ideal scenario for the use of Big Data framework such as Apache Spark. However, the use of Big Data framework in fMRI data analysis is limited due to the difficulty of representing fMRI data in distributed data sets that can then be utilized by such frameworks.

The Resilient Distributed Dataset(RDD) is the primary data abstraction in Apache Spark, which is essentially the fault-toleration collection of records. In memory transformations and actions on RDD support the core of operations on Spark. The use of open source Spark NIfTI Reader library - biananes enables NIfTI image files of fMRI data to be read into RowMatrix RDD for Apache Spark, and thus available to MLlib function calls.

However, some MLlib functions are limited in performance. According to the benchmark results from Intel upon the release of Intel Data Analytics Acceleration Library,[2] on computation of principal component analysis under Spark, DAAL shows a most significant 7x performance boost compared with MLlib. Therefore, in the hope to enhance the analysis process, the use of MLlib is replaced by DAAL with tests to verify performance difference.

To replace MLlib with DAAL, adaptations are needed to make biananes return the desired data structure needed by DAAL. Conversion from RowMatrix to Numeric Table, or more specifically, Homogeneous Numeric Table, is needed.

The last stage of the project is the realization of Full Correlation Matrix Analysis(FCMA) on Spark. With reference to the FCMA toolbox and relevant papers by Princeton University,[3] attempts to implement the 3-stage pipeline of FCMA are made with the updated biananes and use of DAAL.

. Mentors: Pragnesh Patel, Kwai Wong, Junqi Yin

1. rboubela, https://github.com/rboubela/biananes.

2. Intel, https://software.intel.com/en-us/blogs/daal.

3. Yida Wang et al., *Full correlation matrix analysis of fmri data*, technical report ().

II. biananes[4] Update for DAAL

biananes is a library for fMRI data analysis on large datasets running on the Apache Spark framework, by reading fMRI NIfTI files into RowMatrix RDD in Spark MLlib. It is basically written in scala, and compiled with Java compiler.

NIfTI is the most common file type in representing fMRI data, which is basically a collection of arrays representing different dimension of the fMRI image, plus some header information. biananes extracts the data stored in the multi-dimensional array in NIfTI files, and concatenates the data in a continuous row ordered by time volume. The resulting row is in RowMatrix RDD that can then be divided into partitions and distributed for computation on memory of respective worker nodes.

RowMatrix, however, is the data structure in Spark MLlib. The end goal of this project is to test and replace the use of MLlib with Intel DAAL. Therefore, adaptations on biananes is needed. The data format utilized by Intel DAAL in its sample codes for Spark is $JavaPairRDD\langle Integer, HomogenNumericTable\rangle$, where *Integer* takes the index number of the partition, or volume, *HomogenNumericTable* takes the Homogeneous Numeric Table storing the data content.

Homogeneous Numeric Table in DAAL represents features in columns and feature vectors in rows. The adaptation essentially, and theoretically, is to read NIfTI data content into rows in HomogenNumericTable. However, difficulties were encountered when attempting to translate data content in NIfTI to rows in Numeric Table, thus a different approach was used. Figure 1 shows the basic implementation logic of the updated biananes.

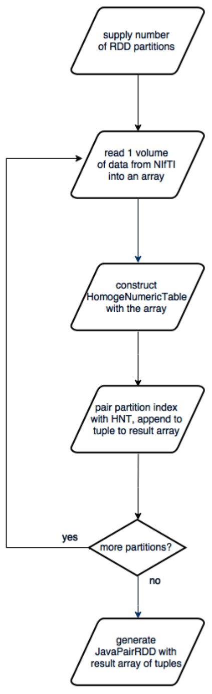

supply number of RDD partitions
read 1 volume of data from NIfTI into an array
construct HomogeNumericTable with the array
pair partition index with HNT, append to tuple to result array
more partitions?
yes
no
generate JavaPairRDD with result array of tuples

Fig. 1. Flowchart of core implementation of first biananes update

Development in later stage of the project required correlation computation by matrix multiplication on Numeric Table input. The construction of Numeric Table in the updated biananes as specified above was then problematic as the multiplication of a single column with its transpose results in the dot product of a single value, which deviates from the expectation of getting a correlation matrix.

4. rboubela,

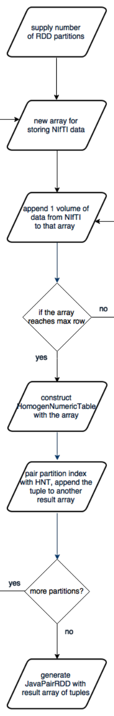

supply number of RDD partitions
new array for storing NIfTI data
append 1 volume of data from NIfTI to that array
if the array reaches max row
no
yes
construct HomogenNumericTable with the array
pair partition index with HNT, append the tuple to another result array
more partitions?
yes
no
generate JavaPairRDD with result array of tuples

Fig. 2. Flowchart of core implementation of second biananes update

A new approach was then developed to generate Numeric Table with multiple rows representing vectors and columns representing features, the implementation logic is illustrated in figure 2.

In this approach, the maximum number of rows in a HomogenNumericTable is limited to the number of partitions, i.e. working nodes. Instead of instantiating a new table from every volume of nifti content, a new volume of nifti content will be appended to an existing table where possible, and a table will only be instantiated when the number of rows has reached the maximum number of rows or when there is no extra partition left.

A. Code Snippet

As mentioned above, the theoretical amendment made to biananes is to translate the construction of rows in RowMatrix to the construction of rows in HomogenNumericTable. However, the test image file contains a total of nearly 500,000 data points per volume, meaning nearly 500,000 columns in 1 single row if the structure of RowMatrix is retained.

This approach was built successfully. But when tested with DAAL sample code on Spark, segmentation fault occurred. An experimental attempt to avoid the problem, corresponding to the flowchart in figure 1, was to construct the table in its transpose manner, i.e. taking rows as a representation of features and columns as a representation of vectors. Basic implementation as in figure 3.

```
while (iter.hasNext) {
  val cur = iter.next
  val curIntger = cur.asInstanceOf[Integer]
  cLib.larray_nifti_read_masked_brick(nii.address, img_file_bc.value, mask_file_bc.value, 1, Array[Int](cur))[Int]

  /*
  * constructor of HomogenNumericTable, taking nfeatures = 1, nVectors = numeber of voxels in mask file
  * nii.toArray is an array containing voxel data in the masked image file
  */
  val hnt = new HomogenNumericTable(dc, nii.toArray, 1, nvoxel_in_mask_bc.value)

  res = res :+ (cur.asInstanceOf(Integer), hnt)
}
```

Fig. 3. Core implementation of single-column construction of *JavaPairRDD*$\langle Integer, HomogenNumericTable\rangle$

Figure 3 shows the basic implementation of the first approach which mimics the construction of rowMatrix in the original biananes code. With this implementation, each volume of data is concatenated into a single column of a HomogenNumericTable.

Sample codes of DAAL, running SVD and QR computation returned results with this updated biananes. This was taken as the implementation for the benchmark tests. However, in later stage of development where matrix-matrix multiplication was needed, new changes were made to biananes.

Figure 4 shows the implementation of the second major update to biananes where multi-row Numeric Table are constructed according to the logic in figure 2.

```
while (iter.hasNext) {
  val cur = iter.next
  val curInteger = cur.asInstanceOf[Integer]
  cLib.larray_nifti_read_masked_brick(nii.address, img_file_bc.value, mask_file_bc.value, 1, Array[Int](cur))[Int]

  val niiArray = nii.toArray
  buf = buf ++ niiArray

  if (curInteger % maxRow == maxRow - 1 || !iter.hasNext) {
    val hnt = new HomogenNumericTable(dc, buf, nvoxel_in_mask_bc.value, curInteger % maxRow + 1)
    res = res :+ (curInteger, hnt)
    buf = null
  }
}
```

Fig. 4. Core implementation of second update of biananes to give multi-row Numeric Table

This approach was built successfully, but when tested with data containing ”fat” matrices - matrices with over 10000 rows, segmentation fault occurred at runtime as with the row-orientated construction of Numeric Table.

Sample codes of DAAL, including QR decomposition and Covariance computation, returned result with the same implementation of biananes on input data with small scale matrices - around 500 rows and columns. The contrary in results with the same update of biananes has speculated doubts on whether DAAL supports the analysis of ”fat” matrices.

B. Setup and Run Walkthrough

biananes is an open source library written in scala with dependencies on a number of other packages and libraries written in C/C++ or Java, including the Spark packages and the NIfTI i/o libraries named niftilib. The dependencies are managed by Apache Maven and thus the compilation and package of the library is done by updating the dependencies in the pom.xml correspondingly and then build with Maven.

The following is the note on how the library was built and used during this project. The platform used throughout the project is the cluster Beacon in NICS.

biananes was built in this project in the following steps,

1) install niftilib i/o library
2) install and load Apache Maven
3) install Intel DAAL
4) install daal.jar locally with maven-install
5) update dependencies in pom.xml
6) update NiftiTools.scala, which contains the implementation of functions in the library
7) use (maven clean &)maven compile & maven package to build the library

Points to note when building the library,

1) daal.jar was installed and linked locally in the maven project, installation and dependencies update are needed when building the library on another platform or user login
2) sparkniftireader.c in bianaes_dir/src/main/C/sparkniftireader/src/ contains functions to be called in NiftiTools.scala, compilation of this code into sparkniftireader.so is needed in advance for the build of biananes.jar
3) lib_so_path in NiftiTools.scala needs to be updated to refer to the correct location sparkniftireader.so
4) Spark version 1.5.2 was used, make sure dependencies involving Spark core in pom.xml are referring to spark 1.5.2 or above

5) inappropriate locale setting might cause error on build, export LC_CTYPE=C if needed

On successful build of biananes, a biananes.jar will result. Import and link the jar file when compile and run the source code containing biananes function calls.

When using the interactive spark-shell or submitting a standalone script with spark-submit, add the following flag:

–jar path_to_biananes.jar

Notes on build and usage are documented in the README.md in the biananes folder uploaded to a svn repository in NICS for the project.

C. Remarks and Evaluation

Discrepancies in results using the same biananes update, i.e. segmentation error on large scale matrix input vs successful return on small scale matrix or tall skinny matrix input might worth investigation on whether it is problem with code implementation or it is that Intel DAAL does not support operations on matrices with huge amount of features.

It is worth noting that the default number of partitions used in biananes is 32, which essentially does not affect the implementation as data with less volumes will just idle the partition without force attempt to have calculation done on those extra partitions.

However, the dimension of Numeric Table instantiated does affect the implementation. Except the case where large scale matrices are concatenated into row-oriented Numeric Table which gives segmentation fault, segmentation error also results when the dimension of Numeric Table exceeds the dimension of data points in the Numeric Table, while a smaller Numeric Table will be constructed and computed without problem, e.g. for a test case with a data containing array size of 1*577, a HomogenNumericTable in the size of 1*578 would return SIGSEGV while a 1*576 table would return results without error.

III. Benchmark Tests

According to benchmark results released by Intel,[5] on computation of Principal Component Analysis on Spark, DAAL shows a max of 7x performance boost compared with MLlib. In the hope of verifying the performance difference and compare such with fMRI data analysis, tests were carried out using DAAL and MLlib on Spark respectively, with the use of the updated biananes as implemented in figure 3.

Figure 5 shows the computation time of DAAL and MLlib on Singular Value Decomposition. Note a volume of data contains a table of 480115 rows and 1 column. To achieve proper scaling effect, for computation with 1 node and 2 nodes, data with 2 volumes are used, and when more nodes are used, data with corresponding volumes are used, e.g. 8-volume data is used for 8-node computation. Also note that for single node computation, it was carried out in local mode while others were carried out in cluster mode.

5. Intel,

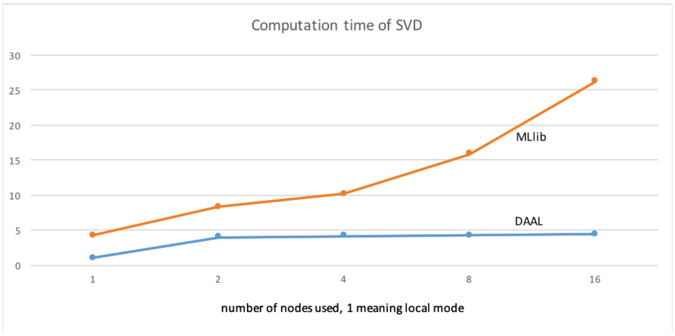


Fig. 5. SVD computation time of DAAL and MLlib

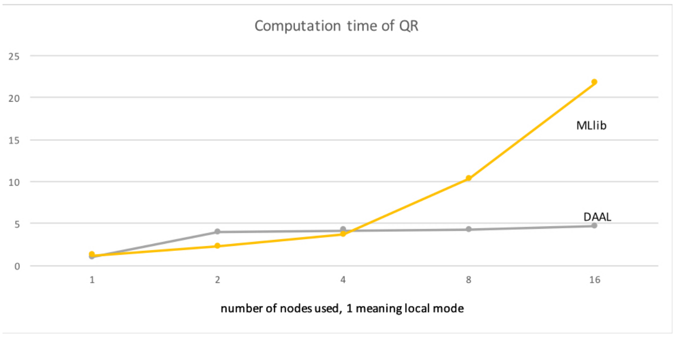


Fig. 6. QR computation time of DAAL and MLlib

Figure 6 shows the computation time of DAAL and MLlib on QR decomposition. Same as that of SVD, variable data size on different nodes are used to achieve proper scaling effect. DAAL has shown performance boost over MLlib on larger data size.

A. Code Snippet

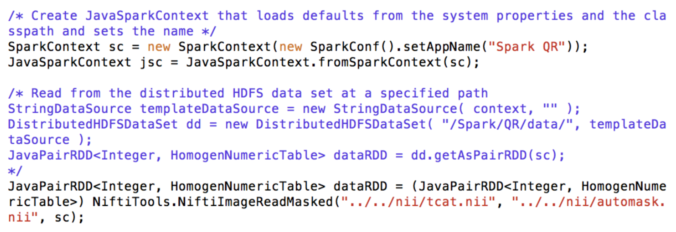
Sample codes in Java of implementations of different algorithms in DAAL come with the installation of DAAL and can be found under the installation directory. Changes were made to the sample codes, mainly to change the way to read data, from the default setting of HDFS under Hadoop and reading of CSV files, to the use of biananes and reading of NIfti files.

The implementations of DAAL algorithms used in the test, i.e. SVD and QR, were left untouched as in the sample implementation from DAAL.

While with MLlib, Java codes were written to repeat the steps of function calls used in the interactive Spark-shell to call SVD and QR computations. The implementations of these analytic computations in MLlib are encapsulated in single functions. The Java codes written basically called SVD and QR with a line of function.

These Java codes had to be written and run with Spark-submit instead of simply calling functions in Scala using Spark-shell because of the hope to align the testing environments, by using Spark-submit with both libraries.

Figure 7 shows the update of the sample code from DAAL. For each computation in DAAL. There is this controller script containing the i/o and main function, which calls the function from another script containing the implementation of the computation. The implementation of SVD is divided



```
/* Create JavaSparkContext that loads defaults from the system properties and the cla
sspath and sets the name */
SparkContext sc = new SparkContext(new SparkConf().setAppName("Spark QR"));
JavaSparkContext jsc = JavaSparkContext.fromSparkContext(sc);

/* Read from the distributed HDFS data set at a specified path
StringDataSource templateDataSource = new StringDataSource( context, "" );
DistributedHDFSDataSet dd = new DistributedHDFSDataSet( "/Spark/QR/data/", templateDa
taSource );
JavaPairRDD<Integer, HomogenNumericTable> dataRDD = dd.getAsPairRDD(sc);
*/
JavaPairRDD<Integer, HomogenNumericTable> dataRDD = (JavaPairRDD<Integer, HomogenNume
ricTable>) NiftiTools.NiftiImageReadMasked("../../nii/tcat.nii", "../../nii/automask.
nii", sc);
```

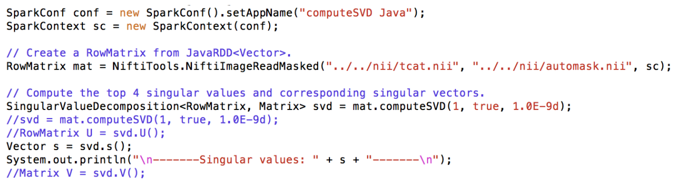
Fig. 7. update to DAAL sample code to read from NIfTI file instead of reading CSv files into HDFS

into 3 steps during distributed computation.[6] The first step is local computation of partial results from the distributed input Numeric Table in each local node, the partial will be kept in the local node for step 3 as well as sent to the master node for step 2; step 2 requires the master node to collect the partial results and split the tables across local nodes to finish up the computation and step 3 is to finish the remaining computation with partial results from step 1 and those acquired from step 2.

QR decomposition shares the same flow of implementation as that of SVD.[7] The implementation scripts of both computations were not touched but remained in their default implementation during the test.

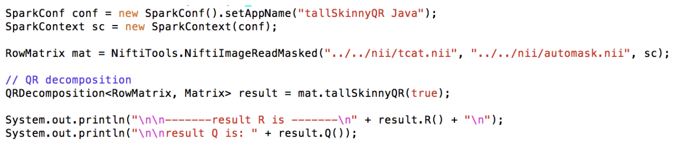
While computations in DAAL are from sample codes which call steps to perform the computations, implementations in MLlib are simply the replicate of Scala commands in Spark-shell in Java codes.

Figure 8 shows the SVD code and figure 9 the QR for MLlib in Java.



```
SparkConf conf = new SparkConf().setAppName("computeSVD Java");
SparkContext sc = new SparkContext(conf);

// Create a RowMatrix from JavaRDD<Vector>.
RowMatrix mat = NiftiTools.NiftiImageReadMasked("../../nii/tcat.nii", "../../nii/automask.nii", sc);

// Compute the top 4 singular values and corresponding singular vectors.
SingularValueDecomposition<RowMatrix, Matrix> svd = mat.computeSVD(1, true, 1.0E-9d);
//svd = mat.computeSVD(1, true, 1.0E-9d);
//RowMatrix U = svd.U();
Vector s = svd.s();
System.out.println("\n-------Singular values: " + s + "-------\n");
//Matrix V = svd.V();
```

Fig. 8. MLlib SVD computation in Java



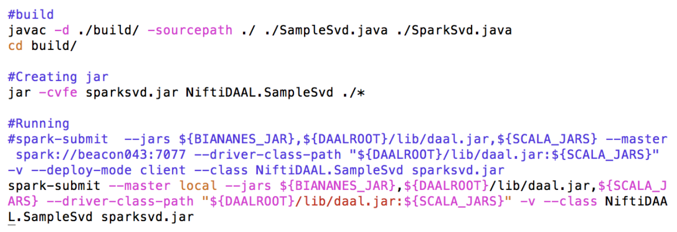
```
SparkConf conf = new SparkConf().setAppName("tallSkinnyQR Java");
SparkContext sc = new SparkContext(conf);

RowMatrix mat = NiftiTools.NiftiImageReadMasked("../../nii/tcat.nii", "../../nii/automask.nii", sc);

// QR decomposition
QRDecomposition<RowMatrix, Matrix> result = mat.tallSkinnyQR(true);

System.out.println("\n\n-------result R is -------\n" + result.R() + "\n");
System.out.println("\n\nresult Q is: " + result.Q());
```

Fig. 9. MLlib QR computation in Java

An additional build script from DAAL has been amended to build the Java codes and run the Spark-submit command. Code snippet of the build script of SVD computation as in figure 10.

The commented out statement under Running is the command to launch in cluster mode, the user can choose between local and cluster by switching the comment tag.

6. Intel, https://software.intel.com/en-us/node/564638.
7. Intel, https://software.intel.com/en-us/node/564644.



```
#build
javac -d ./build/ -sourcepath ./ ./SampleSvd.java ./SparkSvd.java
cd build/

#Creating jar
jar -cvfe sparksvd.jar NiftiDAAL.SampleSvd ./*

#Running
#spark-submit  --jars ${BIANANES_JAR},${DAALROOT}/lib/daal.jar,${SCALA_JARS} --master
 spark://beacon043:7077 --driver-class-path "${DAALROOT}/lib/daal.jar:${SCALA_JARS}"
-v --deploy-mode client --class NiftiDAAL.SampleSvd sparksvd.jar
spark-submit --master local --jars ${BIANANES_JAR},${DAALROOT}/lib/daal.jar,${SCALA_J
ARS} --driver-class-path "${DAALROOT}/lib/daal.jar:${SCALA_JARS}" -v --class NiftiDAA
L.SampleSvd sparksvd.jar
```

Fig. 10. Build & Run script for SVD with DAAL

B. Setup and Run Walkthrough

This section documented the steps taken for the benchmark test. The biananes library file used was built as implemented in figure 3.

Note that for the computation in DAAL, there is the controller script and the implementation script. Changes to the controller script is needed for locating the input file without any changes needed for the implementation file; while for MLlib, changes to locate the input file are updated in the single Java code. To link to the different versions of biananes.jar, the library path is controlled in the build script. The build script serves as both build and run purpose.

1) Update BIANANES_JAR to locate the desired biananes.jar
2) Update SampleSvd.java/SampleQR.java for DAAL or computeSVDmllib.java/tallSkinnyQRmllib.java for MLlib to locate the desired input data file
3) Update the corresponding build scripts to choose from local mode and cluster deployment
4) Load spark/1.5.2
5) Run the corresponding build scripts to build and run the codes

In the test, each and every configuration was run for 10 times. The times obtained are the averaged result from the 10 trials.

C. Remarks and Evaluation

Note that in the test, the structure of Numeric Table used with DAAL is in column orientation, while with MLlib, data are represented by the row in RowMatrix. The difference in structure resulted in difference in the results returned. Figure 11 shows results of SVD with DAAL and figure 12 shows that with MLlib. Despite the difference in U and V returned, the singular values are the same.

While the difference in results returned were neglected, the data used in the test, despite having matrices of around or over 500000 data points, are quite small in size, i.e. around 1MB for the 2-volume data. The computation time was small and trivial compared with the overhead of function setup in the Big Data environment, which could possibly explain the increase in time when the same data file was used in 1-node and 2-node test. With the above said, it is to expect that the comparison results here are too preliminary to serve as an accurate benchmark test on DAAL and MLlib on fMRI data

```
U (2 first vectors from node #0):
0.000
0.000

U (2 first vectors from node #1):
0.000
0.000

Sigma:
153495.057

V:
1.000
```

Fig. 11. Returned result from SVD with DAAL

```
SingularValueDecomposition(null,[153495.0570669953],6.44939004927749E-4
6.910060767083022E-4
7.002194910644121E-4
7.278597341327444E-4
7.46286562844966E-4
7.462865628449658E-4
7.647133915571879E-4
7.554999772010776E-4
7.647133915571875E-4
7.462865628449665E-4
7.647133915571888E-4
7.73926805913299E-4
7.554999772010769E-4
7.462865628449658E-4
7.18646319776635E-4
7.002194910644126E-4
6.54152419283859E-4
6.357255905716372E-4
6.633658336399697E-4
7.278597341327446E-4
7.831402202694085E-4
8.199938776938521E-4
8.660609494744058E-4
9.213414356110694E-4
9.21341435611...
```

Fig. 12. REturned result from SVD with MLlib

analysis. Soon after this project, larger scale data are expected to come and the test shall be conducted again to acquire more meaningful results.

An additional point to note is that, SVD in DAAL tested with fat and short matrices, e.g. 1 row with 200 columns, would fail with Intel MKL error specifying that parameter 1 is incorrect on entry to DORG2R. Lookup to the reference from Intel[8] reveals that ”each data block must have sufficient size”, i.e. number of rows(vectors) must be larger than the number of columns(features), in which case would not be applicable to the data representation in long rows mimicking that of the RowMatrix in MLlib.

IV. FCMA

This section describes the last stage of the project, which is the attempt to migrate the Full Correlation Matrix Analysis toolbox by Princeton University[9] to the Big Data framework Apache Spark with the use of Intel DAAL.

There is the following three stage pipeline in FCMA[10] that this project aims to replicate on the Big Data framework.

1) Correlation computation by matrix-matrix multiplication
2) Normalization of resulted correlation matrix
3) Support Vector Machine to validate result

This FCMA pipeline realizes full correlation matrix analysis in a fairly short period of time, and accuracy can be improved as compared with single-variate analysis. It is originally a toolbox of algorithms and libraries, with its core implementation in parallel computing with MPI. As the nature of FCMA, which deals with a lot of large scale data, is ideal for computation under the Big Data framework, the goal of the project is to employ the concept of the toolbox under Spark.

So far as this project proceeds, development on stage 1 and 3 has started. The intuition was to use matrix-matrix multiplication provided in MKL to develop stage 1. However, DAAL encapsulated the use of MKL that the direct call of matrix-matrix multiplication was not possible.

A. Code Snippet

A workaround to the problem is to make use of Covariance computation. Covariance is provided in the sample codes from DAAL, and result Numeric Table supports the change from type Covariance to type Correlation. By updating the implementation code in figure 14, a multiplication of the input Numeric Table(matrix), with its own transpose is resulted.

```
/* Compute a dense variance-covariance matrix on the master node */
correlationMaster.compute();

/* Finalize computations and retrieve the results */
Result res = correlationMaster.finalizeCompute();

result.correlation = (HomogenNumericTable)res.get(ResultId.correlation);
result.mean = (HomogenNumericTable)res.get(ResultId.mean);
```

Fig. 13. Correlation can be computed by updating return result from ResultId.covariance to ResultId.correlation

Although the code has been updated and test returned results, with the use of column-oriented biananes as implemented in figure 3, the result returned was in dot product of a single value instead of the desired correlation matrix. This was probably due to the multiplication of the column table with its row transpose.

This undesirable result initiated the revision of the first biananes update and has yielded the second update with multi-row table as in figure 4.

V. Conclusion

This project is under the Computational Science for Undergraduate Research Experience programme(CSURE 2016), which is conducted by the Joint Institute for Computational Sciences and led by the University of Tennessee, Knoxville and the Oak Ridge National Laboratory. Due to the time constrain of the programme, this project has not finished.

This project has shown preliminary integration of biananes and Intel DAAL, and in turn shown the start of fMRI data analysis under the Big Data framework.

Future development of the project shall reside with the revision of the biananes updates and most importantly, the review of Intel DAAL. More data are expected to come which would facilitate the testing and reviewing of codes. And the underlying implementation of algorithms and support of its

8. Intel,
9. Wang et al., *Full correlation matrix analysis of fmri data*.
10. Yida Wang, Michael B Anderson, and Ted Willke, “Optimizing Full Correlation Matrix Analysis of fMRI Data on IntelR Xeon PhiTM Coprocessors.”

Numeric Table data structure are crucial to the incorporation of DAAL in fMRI data analysis. The remaining stage of the FCMA pipeline shall be continued to accomplish the project.

VI. Reference

1) Roland N Boubela et al., "Big Data approaches for the analysis of large-scale fMRI data using Apache Spark and GPU processing: A demonstration on resting-state fMRI data from the Human Connectome Project," *Frontiers in neuroscience* 9 (2015)
2) Wang et al., *Full correlation matrix analysis of fmri data*
3) Wang, Anderson, and Willke, "Optimizing Full Correlation Matrix Analysis of fMRI Data on IntelR Xeon PhiTM Coprocessors"